

4 USING THE VDK

In This Chapter

This chapter describes how the VDK implements the general concepts described in Chapter 2, [“Operating System Kernel Concepts”](#). For reference information about the VDK library, see Chapter 6, [“API Reference” on page 6-1](#).

The following sections provide information about the operating system kernel components and operations:

- [“Threads” on page 4-2](#)
- [“Scheduling” on page 4-9](#)
- [“Signals” on page 4-16](#)
- [“Interrupt Service Routines” on page 4-30](#)
- [“Device Drivers” on page 4-35](#)

Threads

When designing an application, you partition it into threads, where each thread is responsible for a piece of the work. Each thread operates independently of the others. A thread performs its duty as if it has its own processor, but can communicate with other threads.

Thread Types

You do not directly define threads; instead, you define thread types. A thread is an instance of a thread type, and is similar to any other user-defined type. In other words, a thread type is a C structure, and every variable of the structures is a thread.

You can create multiple instantiations of the same thread type. Each instantiation of the thread type has its own stack, state, priority, and other local variables. Each thread is individually identified by its `ThreadID`, a handle that can be used to reference that thread in kernel API calls. A thread can gain access to its `ThreadID` by calling `GetThreadID()`. A `ThreadID` is valid for the life of the thread—once a thread is destroyed, the `ThreadID` becomes invalid.

Old `ThreadID`s are eventually reused, but there is significant time between a thread's destruction and the `ThreadID` re-use: other threads have to recognize that the original thread is destroyed.

Thread Parameters

When a thread is created, the system allocates space in the heap to store a data structure that holds the thread-specific parameters. The data structure contains internal information required by the kernel and the thread type specifications provided by the user.

Stack Size

Each thread has its own stack. The full C/C++ run-time model, as specified in the compiler manual, is maintained on a per thread basis. It is your responsibility to assure that each thread has enough room on its stack for all function calls' return addresses and passed parameters appropriate to the particular run-time model, user code structure, use of libraries, etc. Stack overflows do not generate an exception, so an undersized stack has the potential to cause difficulties when reproducing bugs in your system.

Priority

Each thread type specifies a default priority. Threads may change their own (or another thread's) priority dynamically using the `SetPriority()` or `ResetPriority()` functions. Priorities are predefined by the kernel as an enumeration of type `Priority` with a value of `kPriority1` being the highest priority (or the first to be scheduled) in the system. The priority enumeration is set up such that `kPriority1 > kPriority2 > ....` The number of priorities is limited to the processor word size minus two.

Required Thread Functionality

Each thread type is required to have five functions declared and implemented. Default null implementations of all five functions are provided in the templates generated by the VisualDSP++ development environment. The thread's run function is the entry point for the thread. For many thread types, it is the only function in the template that you need to modify. The other functions allocate and free up system resources at appropriate times during the creation and destruction of a thread.

Run Function

The run function—called `Run()` in C++ and `RunFunction()` in C/assembly-implemented threads—is the entry point for a fully constructed thread; `run()` is roughly equivalent to `main()` in a C program. When a thread's run function returns, the thread is moved to the queue of threads

Threads

waiting to free their resources. If the run function never returns, the thread remains running until destroyed.

Error Function

The thread's error function is called by the kernel when an error occurs in an API call made by the thread. The error function passes a description of the error in the form of an enumeration. It also can pass an additional piece of information whose exact definition depends on the error enumeration. A thread's default error-handling behavior destroys the thread. See [page 4-8](#) for more information about error-handling facilities in the VDK.

Create Function

The create function is similar to the constructor. Unlike the constructor, it provides an abstraction used by the kernel API `CreateThread()` to enable dynamic thread creation. The create function is the first function called in the process of constructing a thread; it is also responsible for calling the thread's init function/constructor. Similar to the constructor, the create function executes in the context of the thread that is spawning a new process by calling `CreateThread()`. The thread being constructed does not yet have a run-time context fully established until after these functions complete.

A create function calls the constructor for the thread and ensures that all of the allocation that the thread type required have taken place correctly. If any of the allocation fail, the create function deletes the partially created thread instantiation and returns a null pointer. If the thread has been constructed successfully, the create function returns the pointer to the thread. A create function should not call `DispatchThreadError()` because `CreateThread()` handles error reporting to the calling thread when the create function returns a null pointer.

The create function is exposed completely in C++ source templates. For C or assembly threads, the create function appears only in the thread's header file. If the thread allocates data in `InitFunction()`, you need to

modify the create function in the thread's header to verify that the allocations are successful, and delete the thread if not.

A thread of a certain thread type can be created at boot time by specifying a boot thread of the given thread type in the development environment. Additionally, if the number of threads in the system is known at build time, all the threads can be boot threads.

Init Function/Constructor

The `InitFunction()` (in C/Assembly) and the constructor (in C++) provide a place for a thread to allocate system resources during the dynamic thread creation. A thread uses `malloc` (or `new`) when allocating the thread's local variables. A thread's init function/constructor cannot call any APIs since the function is called from within a different thread's context.

Destructor

The destructor is called by the system when the thread is destroyed. A thread can do this explicitly with a call to `DestroyThread()`. The thread can also be destroyed as a result of an error condition from which it can not recover, or the thread can simply run to completion by reaching the end of its run function and falling out of scope. In all cases, you are responsible for freeing the memory and other system resources that the thread has claimed. Any memory allocated with `malloc` or `new` in the constructor should be released with a corresponding call to `free` or `delete` in the destructor.

A thread is not necessarily destructed when the `DestroyThread()` API is called. `DestroyThread()` takes a parameter that provides a choice of priority when the thread's destructor is called. If the second parameter, `inDestroyNow`, is `FALSE`, the thread is placed in a queue of threads to be cleaned up by the idle thread, and the destructor is called at a priority lower than that of any used threads. While this scheme has many advantages, it works as, in essence, the background garbage collection. This is

Threads

not deterministic and presents no guarantees of when the freed resources are available to other threads. If the `inDestroyNow` argument is passed to `DestroyThread()` with a value of `TRUE`, the destructor is called immediately. This assures that the resources are freed when the function returns, but the destructor is effectively called at the priority of the currently running thread even if a lower-priority thread is being destroyed.

Writing Threads in Different Languages

Thread types may be written in C, C++, or assembly. The choice of language is transparent to the kernel. The development environment generates well-commented skeleton code for all three choices.

One of the key properties of threads is that they are separate instances of the thread type templates—each with a unique local state. The mechanism for allocating, managing, and freeing thread local variables varies from language to language.

C++ Threads

C++ threads have the simplest template code of the three supported languages. User threads are derived classes of the abstract base class `VDK::Thread`. C++ threads have slightly different function names and include a `Create()` function as well a constructor.

Since user thread types are derived classes of the abstract base class `VDK::Thread`, member variables may be added to user thread classes in the header as with any other C++ class. The normal C++ rules for object scope apply so that threads may make use of `public`, `private`, and `static` members. All member variables are thread-specific (or instantiation-specific).

Additionally, calls to VDK APIs in C++ are different from C and assembly calls. All VDK APIs are in the `VDK` namespace. For example, a call to `CreateThread()` in C++ is `VDK::CreateThread()`. We do not recommend exposing the entire VDK namespace in your C++ threads with the `using` keyword.

C and Assembly Threads

Threads written in C rely on a C++ wrapper in their generated header file, but are otherwise common C functions. For this reason, generated C source files do not include the associate header file. C thread function implementations are compiled without the C++ compiler extensions.

In C and assembly programming, the state local to the thread is accessed through a handle (a pointer to a pointer) that is passed as an argument to each of the four user-thread functions. When more than a single word of state is needed, a block of memory is allocated with `malloc()` in the thread type's `InitFunction()`, and the handle is set to point to the new structure.

Each instance of the thread type allocates a unique block of memory, and when a thread of that type is swapped in, the handle references the correct memory reference. Note that, in addition to being available as an argument to all functions of the thread type, the handle can be obtained at any time for the currently running thread using the API function call `GetThreadHandle()`. A thread should *not* call `GetThreadHandle()` in the `InitFunction()` or the `DestroyFunction()`; instead, use the parameter passed to these functions.

Global Variables

VDK applications can use global variables as normal variables. In C or C++, a variable defined in exactly one source file is declared as `extern` in other files in which that variable is used. In assembly, the `.GLOBAL` declaration exposes a variable outside a source file, and the `.EXTERN` declaration resolves a reference to a symbol at link time.

You need to plan carefully how global variables are to be used in a multi-threaded system. Limit access to a single thread or thread class whenever possible to avoid reentrancy problems. Critical and/or unscheduled regions should be used to protect operations on independent

Threads

variables that can potentially leave the system in an undefined state if not completed atomically.

Error Handling Facilities

The VDK includes an error-handling mechanism that allows you to define behavior at the thread type level. Each function call in Chapter 6, “[API Reference](#)”, lists the error codes that may result. For information on the error codes, see “[VersionStruct](#)” on page 5-20.

The assumption underlying the error-handling mechanism in VDK is that all function calls do succeed and, therefore, do not return an explicit error code that you must verify. The VDK’s method differs from common C programming convention in which the return value of every function call must be checked to assure that the call has succeeded without an error. While that model is widely used in conventional systems programming, real-time embedded system function calls rarely, if ever, fail. When an error does occur, the system calls the user-implemented `ErrorFunction()`. You can call the `GetLastThreadError()` API to obtain an enumeration that describes the error condition. You can also call `GetLastThreadError-Value()` to obtain an additional descriptive value whose definition depends on the enumeration. The thread’s `ErrorFunction()` should check if the value returned by `GetLastThreadError()` is one that can be handled intelligently and can perform the appropriate operations. Any enumerated errors that the thread cannot handle must be passed to the default thread error function. For instructions on how to pass an error to the error function, see comments included in the generated thread code.

Scheduling

The scheduler's role is to ensure that the highest-priority ready thread is allowed to run at the earliest possible time. The scheduler is never invoked directly by a thread, but the scheduler's portions are executed whenever a kernel API — called from either a thread or an ISR — changes the highest-priority thread. The scheduler is not invoked during critical or unscheduled regions, but can be invoked immediately at the close of either type of protected region.

Ready Queue

The scheduler relies on an internal data structure known as the *ready queue*. The queue holds references to all threads that are not blocked or sleeping. All threads in the ready queue have all resources needed to run; they are only waiting for processor time. The exception is the currently running thread, which remains in the ready queue during execution.

The data structure is called a queue because it is arranged as a prioritized FIFO buffer. That is, when a thread is moved to the ready queue, it is added as the last entry at its priority. For example, there are four threads in the ready queue at the priority three, five and seven, and an additional thread is made ready with a priority of five (see [Figure 4-1 on page 4-10](#)).

Scheduling

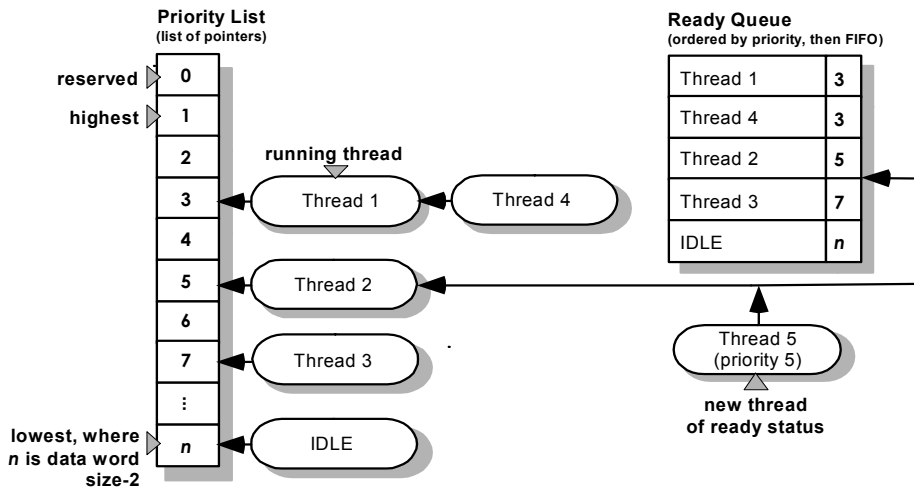


Figure 4-1. Ready Queue

The additional thread is inserted after the old thread with the priority of five, but before the thread with the priority of seven. Threads are added to and removed from the ready queue in a fixed number of cycles regardless of the size of the queue.

Scheduling Methodologies

The VDK always operates as a pre-emptive kernel. However, you can take advantage of a number of modes to expand the options for simpler or more complex scheduling in your applications.

Cooperative Scheduling

Multiple threads may be created at the same priority level. In the simplest scheduling scheme, all threads in the system are given the same priority, and each thread has access to the processor until it manually yields control. This arrangement is called *cooperative multithreading*. When a thread

is ready to defer to the next thread in the FIFO, the thread can do so by calling the `yield()` function, placing the currently running thread at the end of the list. In addition, any system call that causes the currently running thread to block would have a similar result. For example, if a thread depends on a signal that is not currently available, the next thread in the queue at that priority starts running.

Round-Robin Scheduling

Round-robin scheduling, also called time slicing, allows multiple threads with the same priority to be given processor time automatically in fixed duration allotments. In the VDK, priority levels may be designated as round-robin mode at build time and their period specified in system ticks. Threads at that priority should to be run for that duration as measured by the number of timer interrupts. If the thread is pre-empted by a higher-priority thread for a significant amount of time, the time is not subtracted from the time slice. When a thread's round-robin period completes, it is moved to the end of the list of threads at its priority in the ready queue. Note that the round-robin period is subject to jitter when threads at that priority are pre-empted.

Pre-Emptive Scheduling

Full *pre-emptive scheduling*, in which a thread gets processor time as soon as it is placed in the ready queue if it has a higher priority than the running thread, provides more power and flexibility than pure cooperative or round-robin scheduling.

The VDK allows the use of all three paradigms without any modal configuration. For example, a multiple non-time-critical thread can be set to a low priority in the round-robin mode, ensuring that each thread gets processor time without interfering with time-critical threads. Furthermore, a thread can yield the processor at any time, allowing another thread to run. A thread does not need to wait for a timer event to swap the thread out when it has completed the assigned task.

Disabling Scheduling

Sometimes it is necessary to disable the scheduler when making a sequence of API calls. For example, when a thread tries to change the state of more than one signal at a time, the thread can enter an unscheduled region to free all the signals atomically. Unscheduled regions are sections of code that execute without being pre-empted by a higher-priority thread. Note that interrupts are serviced in an unscheduled region, but the same thread runs on return to the thread domain. Unscheduled regions are entered through a call to `PushUnscheduledRegion()`. To exit an unscheduled region, a thread calls `PopUnscheduledRegion()`.

Unscheduled regions (similar to critical regions covered in “[Enabling and Disabling Interrupts](#)” on page 4-30) are implemented with a stack. Using nested critical and unscheduled regions allows you to write code that requires a region without being concerned about the region context when a function is called. For example:

```
void My_UnscheduledFunction()
{
    VDK_PushUnscheduledRegion();
    /* In at least one unscheduled region, but
       this function can be used from any number
       of unscheduled or critical regions */
    /* ... */
    VDK_PopUnscheduledRegion();
}

void MyOtherFunction()
{
    VDK_PushUnscheduledRegion();
    /* ... */
    /* This call adds and removes one unscheduled region */
    My_UnscheduledFunction();
    /* The unscheduled regions are restored here */
    /* ... */
    VDK_PopUnscheduledRegion();
}
```

An additional function for controlling unscheduled regions is `PopNestedUnscheduledRegions()`. This function completely pops the stack of all unscheduled regions. Although the VDK includes `PopNestedUnscheduledRegions()`, applications should use the function infrequently and balance regions correctly.

Entering the Scheduler from API Calls

Since the highest-priority ready thread is the running thread, the scheduler needs to be called only when a higher-priority thread becomes ready. Because a thread interacts with the system through a series of API calls, the times when the number of ready threads changes is well defined. Therefore, a thread invokes the scheduler only when a thread changes the highest-priority ready thread, or leaves an unscheduled region and the highest-priority ready thread has changed. The described VDK's strategy reduces the number of times the scheduler runs, reducing the amount of time spent in the kernel's code.

Entering the Scheduler from Interrupts

In an effort to reduce the number of context switches, interrupt service routines should be written in assembly language. ISRs should communicate to the thread domain through a set of APIs that do not assume any context. Depending on the system state, an ISR API call may require the scheduler being executed. The VDK reserves the lowest-priority interrupt to handle the reschedule process.

If an ISR API call affects the system state, the API raises the lowest-priority interrupt. When the lowest-priority interrupt is scheduled to run by the hardware interrupt dispatcher, the interrupt reduces to subroutine and enters the scheduler. If the interrupted thread is not in an unscheduled region and a higher-priority thread has become ready, the scheduler swaps out the interrupted thread and swaps in the new high-priority ready thread. Additionally, the low-priority software interrupt respects any

Scheduling

unscheduled regions the running thread is in. Yet, the lower-priority interrupt services device drivers, posts periodic semaphores, and moves timed-out threads to the ready queue. When the interrupted thread leaves the unscheduled region, the scheduler is being entered again, and the highest-priority thread ready to run is the new running thread.

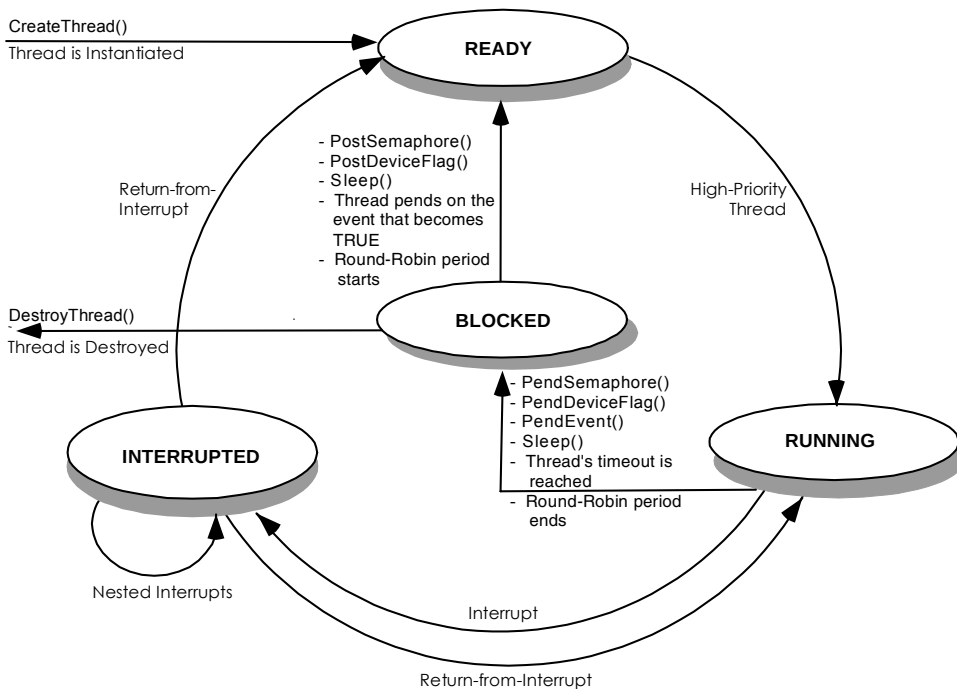


Figure 4-2. Thread State Diagram

Idle Thread

The idle thread is a predefined, automatically created thread that has a priority lower than that of any user threads. Thus, when there are no user threads in the ready queue, the idle thread runs. The only substantial work performed by the idle thread is the freeing of resources of threads that

have been destroyed. In other words, the idle thread handles destruction of threads that were passed to `DestroyThread()` with a value of `FALSE` for `inDestroyNow`.

The time spent in the threads other than the idle thread is shown plotted as a percentage over time on the **Load** tab of the **State History** window in VisualDSP++. See [page 3-44](#) for more information about the State History window.

Signals

Threads have three different methods for communication and synchronization:

- Semaphores
- Events
- Device Flags

Each communication method has a different behavior and use. A thread pends on any of the three types of signals, and if a signal is unavailable, the thread blocks until the signal becomes available or (optionally) a timeout is reached.

Semaphores

Semaphores are protocol mechanisms offered by most operating systems. Semaphores are used to:

- Control access to a shared resource
- Signal a certain system occurrence
- Allow two threads to synchronize
- Schedule periodic execution of threads

The number and initial state of semaphores is set up when your project is built.

Behavior of Semaphores

A semaphore is a token that a thread acquires so that the thread can continue execution. If the thread pends on the semaphore and it is not in use by another thread, the semaphore is acquired, and the thread continues normal execution. If the semaphore is already in use by another thread, the thread trying to acquire (pend) on the semaphore blocks until the semaphore is available, or the specified timeout occurs. If the semaphore does not become available in the time specified, the thread continues execution in its error function.

Semaphores are global structures that are accessible to all threads in the system. Threads of different types and priorities can pend on a semaphore. When the semaphore is posted, the thread with the highest priority that has been waiting the longest is moved to the ready queue. Additionally, unlike many operating systems, VDK semaphores are not owned. In other words, any thread is allowed to post a semaphore (make it available)—not just the thread that has the semaphore. If a thread has requested (pended) and received the semaphore, and the thread is destroyed, the semaphore is not released.

Besides operating as a flag between threads, a semaphore can be set up to be periodic. A periodic semaphore is posted by the kernel every n ticks, where n is the period of the semaphore. Periodic semaphores can be used to ensure that a thread is run at regular intervals.

Thread's Interaction with Semaphores

Threads interact with semaphores through the set of semaphore APIs. The functions allow a thread to pend on a semaphore, post a semaphore, get a semaphore's value, and add or remove a semaphore from the periodic queue.

Pending on a Semaphore

Figure 4-3 illustrates the process of pending on a semaphore.

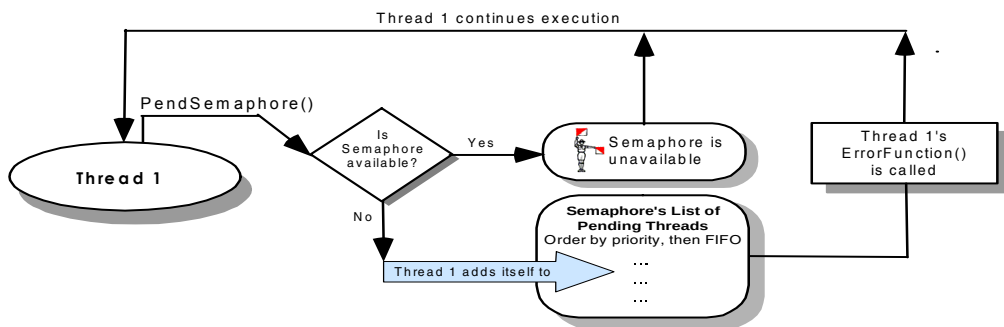


Figure 4-3. Pending on a Semaphore

Threads can pend on a semaphore with a call to `PendSemaphore()`. When a thread calls `PendSemaphore()`, it either acquires the semaphore and continues execution, or blocks until the semaphore is available or the specified timeout occurs. If the semaphore becomes available before the timeout occurs, the thread continues execution; otherwise, the thread's error function is called and the thread continues execution. You should not call `PendSemaphore()` within an unscheduled or critical region because the call may activate the scheduler. Pending with a timeout of zero on a semaphore pends without timeout.

Posting a Semaphore

Semaphores can be posted from two different scheduling domains: the thread domain and the interrupt domain. Posting a semaphore moves the highest-priority thread from the semaphore's list of pending threads to the ready queue. All other threads are left blocked on the semaphore until their timeout occurs, or the semaphore becomes available for them.

Posting from the Thread Domain. [Figure 4-4](#) and [Figure 4-5 on page 4-20](#) illustrate the process of posting semaphores from the thread domain.

A thread can post a semaphore with a call to the `PostSemaphore()` API. If a thread calls `PostSemaphore()` from within a scheduled region (see [Figure 4-4](#)), and a higher-priority thread is moved to the ready queue, the thread calling `PostSemaphore()` is context switched out.

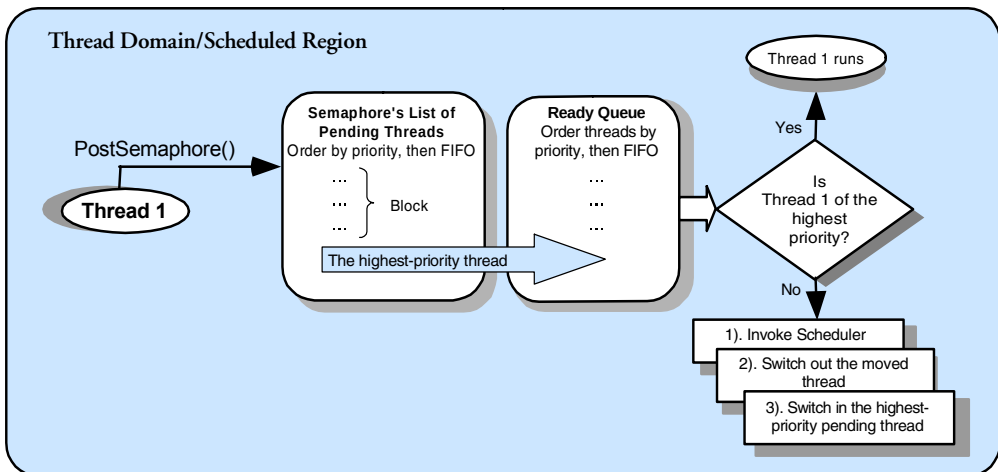


Figure 4-4. Thread Domain/Scheduled Region: Posting a Semaphore

If a thread calls `PostSemaphore()` from within an unscheduled region, where the scheduler is disabled, the highest-priority thread moved to the ready queue runs (see [Figure 4-5](#)).

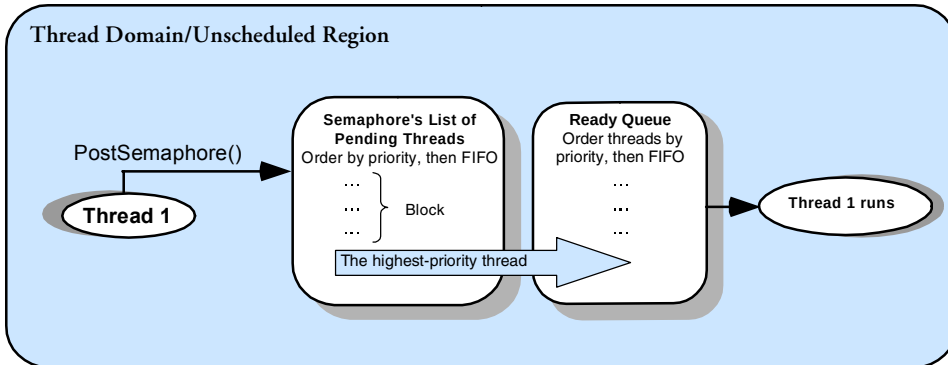


Figure 4-5. Thread Domain/Unscheduled Region: Posting a Semaphore

Posting from the Interrupt Domain. Interrupt subroutines can also post semaphores. [Figure 4-6](#) illustrates the process of posting a semaphore from the interrupt domain.

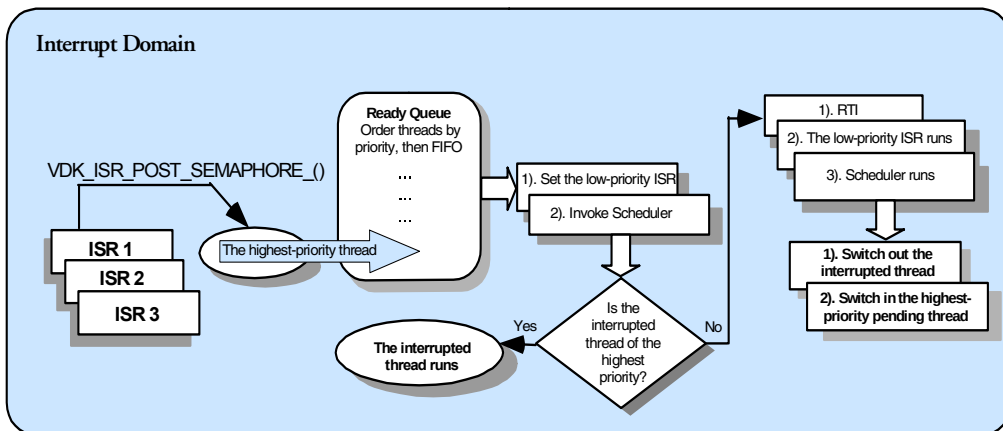


Figure 4-6. Interrupt Domain: Posting a Semaphore

An ISR posts a semaphore by calling the `VDK_ISR_POST_SEMAPHORE()` macro. The macro moves the highest-priority thread to the ready queue and sets the low-priority software interrupt if a call to the scheduler is required. When the ISR completes execution, and the low-priority software interrupt is run, the scheduler is run. If the interrupted thread is in a scheduled region, and a higher-priority thread becomes ready, the interrupted thread is switched out and the new thread is switched in.

Periodic Semaphores

Semaphores can also be used to schedule periodic threads. The semaphore is posted every n ticks (where n is the semaphore's period). A thread can then pend on the semaphore and be scheduled to run every time the semaphore is posted. A periodic semaphore does not guarantee that the thread pending on the semaphore is the highest-priority scheduled to run, or that scheduling is enabled. All that is guaranteed is that the semaphore is posted, and the highest-priority thread pending on that semaphore moves to the ready queue.

Periodic semaphores are posted by the kernel during the timer interrupt at system tick boundaries. Periodic semaphores can also be posted at any time with a call to `PostSemaphore()` or `VDK_ISR_POST_SEMAPHORE()`. Calls to these functions do not affect the periodic nature of the semaphore.

Events and Event Bits

Events and event bits are signals used to regulate thread execution based on the state of the system. An event bit is used to signal that a certain system element is in a specified state. An event is a Boolean operation performed on the state of all event bits. When the Boolean combination of event bits is such that the event evaluates to `TRUE`, all threads that are pending on the event are moved to the ready queue and the event remains `TRUE`. Any thread that pends on an event that evaluates as `true` does not block, but when event bits have changed causing the event to evaluate as `FALSE`, any thread that pends on that event blocks.

Signals

Due to the event and event bit data structures, many scheduling operations associated with them are non-deterministic. Because of this, different VDK libraries are linked in at build time. A library that includes event and event bit code is linked in if the development environment defines any events; otherwise, a library without events is linked in.

The number of events and event bits is limited to a processor's word size minus one. For example, on a sixteen-bit architecture, there can only be fifteen events and event bits; and on a thirty-two-bit architecture, there can be thirty-one of each.

Behavior of Events

Each event maintains the `VDK_EventData` data structure that encapsulates all the information used to calculate an event's value:

```
typedef struct
{
    bool                matchAll;
    VDK_Bitfield        mask;
    VDK_Bitfield        values;
} VDK_EventData;
```

When setting up an event, you configure a flag describing how to treat a mask and target value:

- `matchAll`: `TRUE` when an event must have an exact match on all of the masked bits. `FALSE` if a match on any value results in the event re-calculating to `TRUE`.
- `mask`: The event bits that the event calculation is based on.
- `values`: The target values for the event bits masked with the `mask` field of the `VDK_EventData` structure.

Unlike semaphores, events are `TRUE` whenever their conditions are `TRUE`, and all threads pending on the event are moved to the ready queue. If a thread pends on an event that is already `TRUE`, the thread continues to run,

and the scheduler is not called. Like a semaphore, a thread pending on an event that is not `TRUE` blocks until the event becomes true, or the thread's timeout is reached. Pending with a timeout of zero on an event pends without timeout.

Global State of Event Bits

The state of all the event bits is stored in a global variable. When a user sets or clears an event bit, the corresponding bit number in the global word is changed. If toggling the event bit affects any events, that event is recalculated. This happens either during the call to `SetEventBit()` or `ClearEventBit()` (if called within a scheduled region), or the next time the scheduler is enabled (with a call to `PopUnscheduledRegion()`).

Event Calculation

To understand how events use event bits, see the following examples.

Example 1. Calculation for an 'all' event.

4	3	2	1	0	event bit number	
0	1	0	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is FALSE because the global event bit 2 is not the target value.

Example 2. Calculation for an 'all' event.

4	3	2	1	0	event bit number	
0	1	1	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is TRUE.

Signals

Example 3. Calculation for an ‘any’ event.

4	3	2	1	0	event	bit number
0	1	0	1	0	<—	bit value
0	1	1	0	1	<—	mask
0	1	1	0	0	<—	target value

Event is TRUE since bits 0 and 3 of the target and global match.

Example 4. Calculation for an ‘any’ event.

4	3	2	1	0	event	bit number
0	1	0	1	1	<—	bit value
0	1	1	0	1	<—	mask
0	0	1	0	0	<—	target value

Event is FALSE since bits 0, 2, and 3 do not match.

Effect of Unscheduled Regions on Event Calculation

Every time an event bit is set or cleared, the scheduler is entered to recalculate all dependent event values. By entering an unscheduled region, you can toggle multiple event bits without triggering spurious event calculations that could result in erroneous system conditions. Consider the following code:

```
// Code that accidentally triggers Event1 trying to set up Event2.  
// Assume the prior event bit state = 0x00.  
  
VDK_EventData data1 = { true, 0x3, 0x1 };  
VDK_EventData data2 = { true, 0x3, 0x3 };  
VDK_LoadEvent(kEvent1, data1);  
VDK_LoadEvent(kEvent2, data2);  
VDK_SetEventBit(kEventBit1); // will trigger Event1 by accident  
VDK_SetEventBit(kEventBit2); // Event1 is false, Event2 is true
```

Whenever you toggle multiple event bits, you should enter an unscheduled region to avoid the above loopholes. For example, to fix the above accidental triggering of Event1 in the above code, use the following code:

```

VDK_PushUnscheduledRegion();
VDK_SetEventBit(kEventBit1); // Event1 has not been triggered *
VDK_SetEventBit(kEventBit2); // Event1 is false, Event2 is true
VDK_PopUnscheduledRegion();

```

Thread's Interaction with Events

Threads interact with events by pending on events, setting or clearing event bits, and by loading a new `VDK_EventData` into a given event.

Pending on an Event

Like semaphores, threads can pend on an event's condition becoming `TRUE` with a timeout. [Figure 4-7](#) illustrates the process of pending on an event.

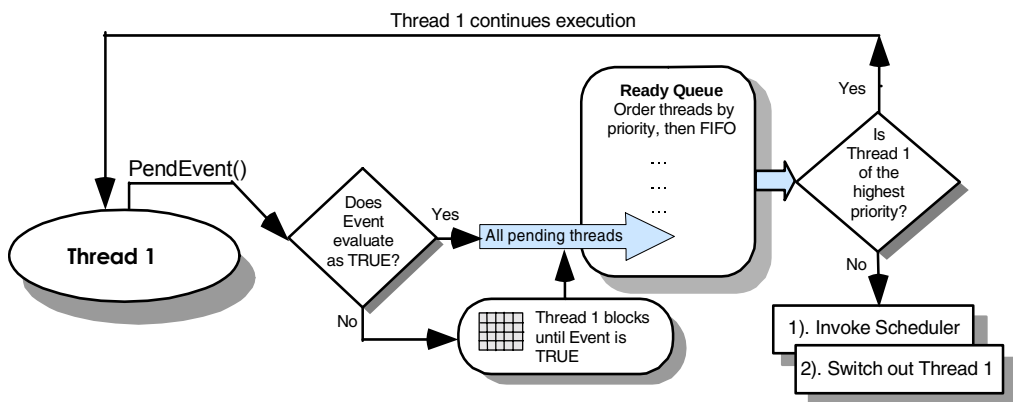


Figure 4-7. Pending on an Event

Signals

A thread calls `PendEvent()` and specifies the timeout. If the event becomes `TRUE` before the timeout is reached, the thread (and all other threads pending on the event) is moved to the ready queue. Calling `PendEvent()` with a timeout of zero means that the thread is willing to wait indefinitely.

Setting or Clearing of Event Bits

Changing the status of the event bits can be accomplished in both the interrupt domain and the thread domain. Each domain results in slightly different results.

From the Thread Domain. Figure 4-8 illustrates the process of setting or clearing of an event bit from the thread domain.

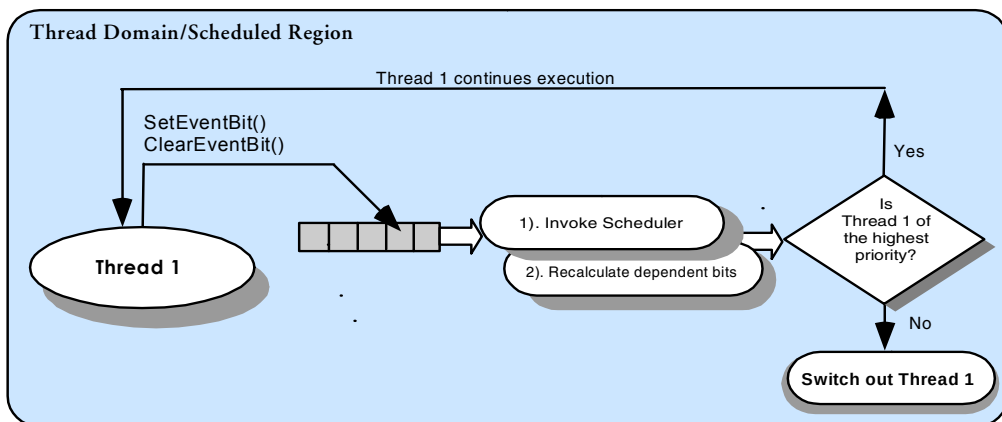


Figure 4-8. Thread Domain: Setting or Clearing an Event Bit

A thread can set an event bit by calling `SetEventBit()` and clear it by calling `ClearEventBit()`. Calling either from within a scheduled region recalculates all events that depend on the event bit and can result in a higher-priority thread being context switched in.

From the Interrupt Domain. Figure 4-9 illustrates the process of setting or clearing of an event bit from the interrupt domain.

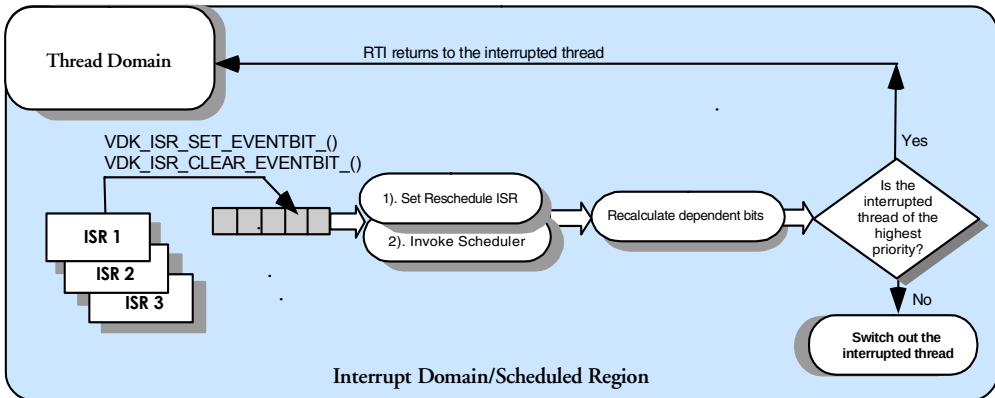


Figure 4-9. Interrupt Domain: Setting or Clearing an Event Bit

An Interrupt Service Routine can call `VDK_ISR_SET_EVENTBIT_()` and `VDK_ISR_CLEAR_EVENTBIT_()` to change an event bit values and, possibly, free a new thread to run. Calling these macros *does not* result in a recalculation of the events, but the low-priority software interrupt is set and the scheduler entered. If the interrupted thread is in a scheduled region, an event recalculation takes place, and can cause a higher-priority thread to be context switched in. If an ISR sets or clears multiple event bits, the calls do not need to be protected with an unscheduled region (since there is no thread scheduling in the interrupt domain); for example:

```

/* The following two ISR calls do not need to be protected: */
VDK_ISR_SET_EVENTBIT_(kEventBit1);
VDK_ISR_SET_EVENTBIT_(kEventBit2);

```

Signals

Loading New Event Data into an Event

From the thread scheduling domain, a thread can get the `VDK_EventData` associated with an event with the `GetEventData()` API. Additionally, a thread can change the `VDK_EventData` with the `LoadEvent()` API. A call to `LoadEvent()` causes a recalculation of the event's value. If a higher-priority thread becomes ready because of the call, it starts running if the scheduler is enabled.

Device Flags

Because of the special nature of device drivers, most require synchronization methods that are similar to that provided by events and semaphores, but with different operation. Device flags are created to satisfy the specific circumstances device drivers might require. Much of their behavior cannot be fully explained without an introduction to device drivers, which are covered extensively in [“Device Drivers” on page 4-35](#).

Behavior of Device Flags

Like events and semaphores, a thread can pend on a device flag, but unlike semaphores and events, a device flag is always `FALSE`. A thread pending on a device flag immediately blocks. When a device flag is posted, all threads pending on it are moved to the ready queue.

Device flags are used to communicate to any number of threads that a device has entered a particular state. For example, assume that multiple threads are waiting for a new data buffer to become available from an A/D converter device. While neither a semaphore nor an event can correctly represent this state, a device flag's behavior can encapsulate this system state.

Thread's Interaction with Device Flags

A thread accesses a device flag through two APIs: `PendDeviceFlag()` and `PostDeviceFlag()`. Unlike most APIs that can cause a thread to block, `PendDeviceFlag()` *must* be called from within a critical region. `PendDeviceFlag()` is set up this way because of the nature of device drivers. See section “[Device Drivers](#)” on page 4-35 for a more information about device flags and device drivers.

Interrupt Service Routines

Unlike the Analog Devices standard C implementation of interrupts (using `signal.h`), all VDK interrupts are written in assembly. The VDK encourages users to write interrupts in assembly by giving hand-optimized macros to communicate between the interrupt domain and the thread domain. All calculations should take place in the thread domain, and interrupts should be short routines that post semaphores, change event bit values, activate device drivers (which are written in C), and drop tags in the history buffer.

Enabling and Disabling Interrupts

Each DSP architecture has a slightly different mechanism for masking and unmasking interrupts. Some architectures require that the state of the interrupt mask be saved to memory before servicing an interrupt or an exception, and the mask be manually restored before returning. Since the kernel installs interrupts (and exception handlers on some architectures), directly writing to the interrupt mask register may produce unintended results. Therefore, VDK provides a simple and platform-independent API to simplify access to the interrupt mask.

A call to `VDK::GetInterruptMask()` returns the actual value of the interrupt mask, even if it has been saved temporarily by the kernel in private storage. Likewise, `VDK::SetInterruptMaskBits()` and `VDK::ClearInterruptMaskBits()` set and clear bits in the interrupt mask in a robust and safe manner. Interrupt levels with their corresponding bits set in the interrupt mask are enabled when interrupts are globally enabled. See the *Hardware Reference Specification* for the processor you are using for more information about the interrupt mask.

VDK also presents a standard way of turning interrupts on and off globally. Like unscheduled regions, the VDK supports critical regions where interrupts are disabled. A call to `PushCriticalRegion()` turns off interrupts, and a call to `PopCriticalRegion()` re-enables interrupts. These API

calls implement a stack-style interface as described in [“Protected Regions” on page 2-9](#). Users are discouraged from turning interrupts off for long sections of code since this increases interrupt latency.

Interrupt Architecture

Interrupt handling can be set up in two ways: support C functions and install them as handlers, or support small assembly ISRs that set flags that are handled in threads or device drivers (which are written in a high-level language). Analog Devices standard C model for interrupts uses `signal.h` to install and remove signal (interrupt) handlers that can be written in C. The problem with this method is that the interrupt space requires a C run-time context, and any time an interrupt occurs, the system must perform a complete context save/restore. The `signal.h` method also increases interrupt latency since every context save/call/restore interrupt must be contained within a critical region.

VDK's interrupt architecture does not support the `signal.h` strategy for handling interrupts. VDK interrupts should be written in assembly, and their body should set some flags that communicate back to the thread or device driver domain. This architecture reduces the number of context saves/restores required, decreases interrupt latency, and still keeps as much code as possible in a high language.

The lightweight nature of ISRs also encourages the use of interrupt nesting to further reduce latency. VDK enables interrupt nesting by default on processors that support it. On processors that support interrupt nesting, the VDK turns it on by default.

Interrupt Service Routines

Vector Table

VDK installs a common header in every entry in the interrupt table. The header disables interrupts and jumps to the interrupt handler. Interrupts are disabled in the header so that you can depend on having access to global data structures at the beginning of their handler. You must remember to re-enable interrupts before executing an RTI.

The VDK reserves two interrupts: the timer interrupt and the lowest-priority interrupt. For a discussion about the timer interrupt, see [“Timer ISR” on page 4-34](#). For information about the lowest-priority interrupt, see [“Reschedule ISR” on page 4-34](#).

Global Data

Often ISRs need to communicate data back and forth to the thread domain besides semaphores, event bits, and device driver activations. ISRs can use global variables to get data to the thread domain, but you must remember to wrap any access to or from that global data in a critical region and to declare the variable as `volatile` (in C/C++). For example, consider the following:

```
// MY_ISR.asm

.extern _my_global_integer;
<REG> = data;
DM(_my_global_integer) = <REG>;
// finish up the ISR, enable interrupts, and RTI.
```

And in the thread domain:

```
/* My_C_Thread.c */

volatile int my_global_integer;
/* Access the global ISR data */
VDK_PushCriticalRegion();
If (my_global_integer == 2)
my_global_integer = 3;
VDK_PopCriticalRegion();
```

Communication with the Thread Domain

The VDK supplies a set of macros that can be used to communicate system state to the thread domain. Since these macros are called from the interrupt domain, they make no assumptions about processor state, available registers, or parameters. In other words, the ISR macros can be called without consideration of saving state or having processor state trampled during a call. Take for example, the following three equivalent

`VDK_ISR_POST_SEMAPHORE_()` calls:

```
.VAR/DATA    semaphore_id;

// Pass the value directly
VDK_ISR_POST_SEMAPHORE_(kSemaphore1);

// Pass the value in a register
<REG> = kSemaphore1;
VDK_ISR_POST_SEMAPHORE_(<REG>);
// <REG> was not trampled

// Post the semaphore one last time using a DM
DM(semaphore_id) = <REG>;
VDK_ISR_POST_SEMAPHORE_(DM(semaphore_id));
```

Additionally, no condition codes are affected by the ISR macros, no assumptions are made about having space on any processor stacks, and all VDK internal data structures are maintained.

Most ISR macros raise the low-priority software interrupt if thread domain scheduling is required after all other interrupts are serviced. For a discussion of the low-priority software interrupt, see section [“Reschedule ISR”](#). Refer to [“Processor Specific Notes” on page A-1](#) for additional information about ISR APIs.

Within the interrupt domain, every effort should be made to enable interrupt nesting. Nesting is always disabled when an ISR begins. However, leaving it disabled is analogous to staying in an unscheduled region in the thread domain; other ISRs are prevented from executing, even if they have

Interrupt Service Routines

higher priority. Allowing nested interrupts potentially lowers interrupt latency for high priority interrupts.

Timer ISR

The VDK reserves the timer interrupt. The timer is used to calculate round-robin times, sleeping threads' time to keep sleeping, and periodic semaphores. One VDK tick is defined as the time between timer interrupts and is the finest resolution measure of time in the kernel. The timer interrupt can cause a low-priority software interrupt (see [“Reschedule ISR”](#)).

Reschedule ISR

The VDK designates the lowest-priority interrupt that is not tied to a hardware device as the reschedule ISR. This ISR handles housekeeping when an interrupt causes a system state change that can result in a new high-priority thread becoming ready. If a new thread is ready and the system is in a scheduled region, the software ISR saves off the context of the current thread and switches to the new thread. If an interrupt has activated a device driver, the low-priority software interrupt calls the dispatch function for the device driver. [For more information, see “Dispatch Function” on page 4-40.](#)

On systems where the lowest-priority non-hardware-tied interrupt is not the lowest-priority interrupt, all lower-priority interrupts must run with interrupts turned off for their entire duration. Failure to do so may result in undefined behavior.

Device Drivers

The role of a device driver is to abstract the details of the hardware implementation from the software designer. For example, a software engineer designing a finite impulse response (FIR) filter does not need to understand the intricacies of the converters, and is able to concentrate on the FIR algorithm. The software can then be reused on different platforms, where the hardware interface differs.

The Communication Manager controls device drivers in the VDK. Using the Communication Manager APIs, you can maintain the abstraction layers between device drivers, interrupt service routines, and executing threads. This section details how the Communication Manager is organized.

Execution

Device drivers and interrupt service routines are tied very closely together. Typically, DSP developers prefer to keep as much time critical code in assembly as possible. The Communication Manager is designed such that you can keep interrupt routines in assembly (the time critical pieces), and interface and resource management for the device in a high-level language without sacrificing speed. The Communication Manager attempts to keep the number of context switches to a minimum, to execute management code at reasonable times, and to preserve the order of priorities of running threads when a thread uses a device. However, you need thoroughly understand the architecture of the Communication Manager to write your device driver.

There is only one interface to a device driver—through a dispatch function. The dispatch function is called when the device is initialized, when a thread uses a device (open/close, read/write, control), or when an interrupt service routine transfers data to or from the device. The dispatch function handles the request and returns. Device drivers should *not* block (pend) when servicing an initialize request or a request for more data by

Device Drivers

an interrupt service routine. However, a device driver can block when servicing a thread request and the relevant resource is not ready or available. Device driver initialization and ISR requests are handled within critical regions enforced by the kernel, so their execution does not have to be reentrant, but a thread level request must protect global variables within critical or unscheduled regions.

Parallel Scheduling Domains

This section focuses on a unique role of device drivers in the VDK architecture. Understanding device drivers requires some understanding of the time and method by which device driver code is invoked. VDK applications may be factored into two *domains*, referred to as the thread domain and the ISR domain (see [Figure 4-10](#)). This distinction is not an arbitrary or unnecessary abstraction. The hardware architecture of the processor as well as the software architecture of the kernel reinforces this notion. You should consider this distinction when you are designing your application and apportioning your code.

Threads are scheduled based on their priority and the order in which they are placed in the ready queue. The scheduling portion of the kernel is responsible for selecting the thread to run. However, the scheduler does not have complete control over the processor. It may be pre-empted by a parallel and higher-priority scheduler: the interrupt and exception hardware. While interrupts or exceptions are being serviced, thread priorities are temporarily moot. The position of threads in the ready queue becomes significant again only when the hardware relinquishes control back to the software-based scheduler.

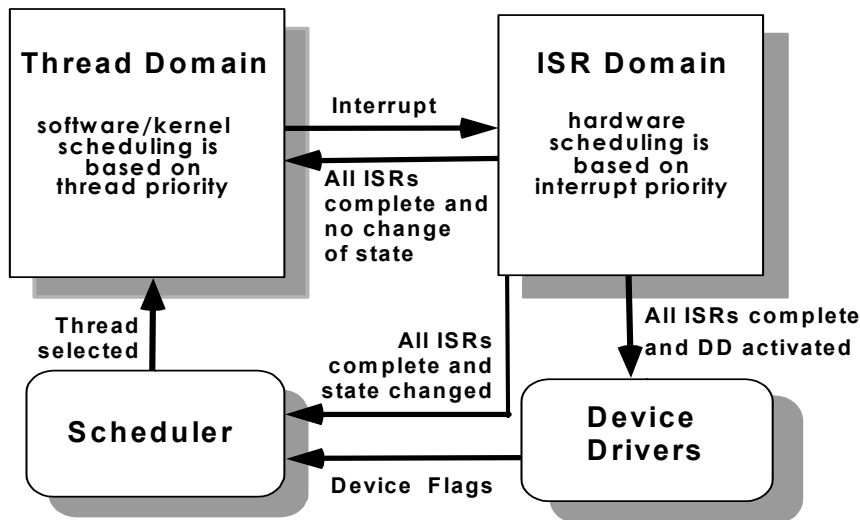


Figure 4-10. Parallel Scheduling Domains

Each of the domains has strengths and weaknesses that dictate the type of code suitable to be executed in that environment. The scheduler in the thread domain is invoked when threads are moved to or from the ready queue. Threads each have their own stack and may be written in a high-level language. Threads always execute in "normal mode" or "user mode" (if the processor make this distinction). Threads implement algorithms and are allotted processor time based on the completion of higher-priority activity.

In contrast, scheduling in the interrupt domain has the highest system-wide priority. Any "ready" ISR takes precedence over any ready thread (outside critical regions), and this form of scheduling is implemented in hardware. ISRs are always written in assembly and must manually restore any registers they use. ISRs execute in "supervisor" or "kernel mode" (if the processor make this distinction). ISRs respond to

Device Drivers

asynchronous peripherals at the lowest level only. The routine should perform only activities that are so time-critical that data would be lost if the code were not executed as soon as possible. All other activity should occur under the control of the kernel's scheduler based on priority.

Transferring from the thread domain to the interrupt domain is simple and automatic, but returning to the thread domain can be much more laborious. If the ready queue is not changed while in the interrupt domain, then the scheduler need not run when it regains control of the system. The interrupted thread resumes execution immediately. If the ready queue has changed, the scheduler must further determine whether the highest-priority thread has changed. If it has changed, the scheduler must initiate a context switch.

Device drivers fill the gap between the two scheduling domains. They are neither thread code nor ISR code, and they are not directly scheduled by either the kernel or the interrupt controller. On processors that make the distinction, they run partly in user mode and partly in supervisor mode. Device drivers are implemented as a single function, but that function is invoked from many different places. Device drivers are typically written in a high-level language and run on the stack of the currently running thread. However, they are not "owned" by any thread, and may be used by many threads concurrently.

Using Device Drivers

From the point of view of a thread, there are five functional interfaces to device drivers: `VDK::OpenDevice()`, `VDK::CloseDevice()`, `VDK::SyncRead()`, `VDK::SyncWrite()`, and `VDK::DeviceIOctl()`. The names of the functions are self-explanatory since threads mostly treat device drivers as black boxes. [Figure 4-11](#) illustrates device drivers' interface. A thread uses a device by opening it, reading and/or writing to it, and closing it. The `VDK::DeviceIOctl()` function is used for sending device-specific control information messages. Each API is a standard C/C++ function call that runs on the stack of the calling thread and returns when the function com-

pletes. However, when the device driver does not have a needed resource, one of these functions may cause the thread to be removed from the ready queue and block on a signal, similar to a semaphore or an event, called a device flag.

Interrupt service routines have only one API call relating to device drivers: `VDK_ISR_ACTIVATE_DEVICE_DRIVER()`. This macro is not a function call, and program flow does not transfer from the ISR to the device driver and back. Rather, the macro sets a flag indicating that the device driver's "activate" routine should execute after all interrupts have been serviced.

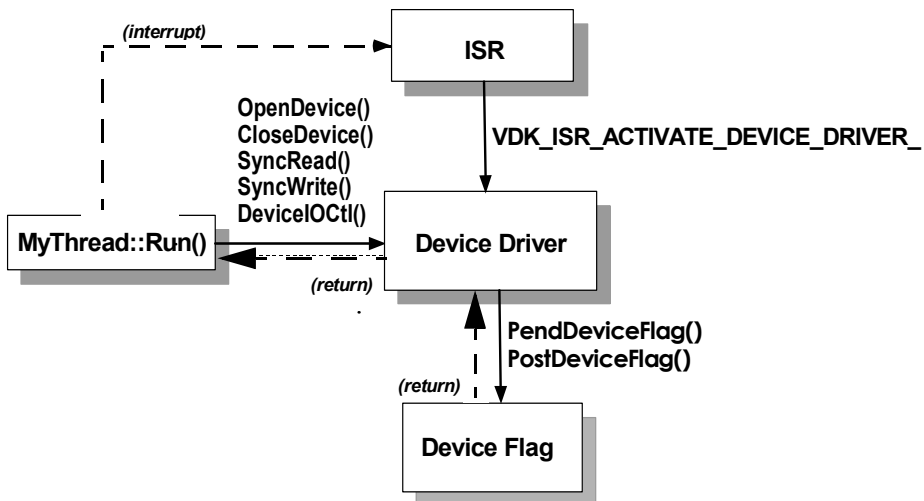


Figure 4-11. Device Driver APIs

The remaining two API functions, `VDK::PendDeviceFlag()` and `VDK::PostDeviceFlag()`, are called only from within the device driver itself. For example, a call from a thread to `VDK::SyncRead()` might cause the device driver to call `VDK::PendDeviceFlag()` if there is no data currently available. This would cause the thread to block until the device flag is posted by another code fragment within the device driver that is providing the data.

Device Drivers

As another example, when an interrupt occurs because an incoming data buffer is full, the ISR might move a pointer so that the device begins filling an empty buffer before calling `VDK_ISR_ACTIVATE_DEVICE_DRIVER()`. The device driver's activate routine may respond by posting a device flag and moving a thread to the ready queue so that it can be scheduled to process the new data.

Dispatch Function

The device driver's only entry point is the dispatch function. The dispatch function takes two parameters and returns a `void*` (the return value depends on the input values). Below is a declaration of a device dispatch function:

```
void* MyDeviceDispatch(VDK_DeviceDispatchID inCode,  
                      VDK_DispatchUnion   inData);
```

The first parameter is an enumeration that specifies why the dispatch function has been called:

```
enum VDK_DeviceDispatchID  
{  
    VDK_kDD_Init,  
    VDK_kDD_Activate,  
    VDK_kDD_Open,  
    VDK_kDD_Close,  
    VDK_kDD_SyncRead,  
    VDK_kDD_SyncWrite,  
    VDK_kDD_IOCTL  
};
```

The second parameter is a union whose value depends on the enumeration value:

```
union DispatchUnion
{
    struct OpenClose_t
    {
        void      **dataH;
        char      *flags; /* used for kDD_Open only */
    };
    struct ReadWrite_t
    {
        void      **dataH;
        VDK_Ticks timeout;
        unsigned int dataSize;
        int       *data;
    };

    struct IOCntl_t
    {
        void      **dataH;
        VDK_Ticks timeout;
        int       command;
        char      *parameters;
    };
};
```

The values in the union are only valid when the enumeration specifies that the dispatch function has been called from the thread domain (kDD_Open, kDD_Close, kDD_SyncRead, kDD_SyncWrite, kDD_IOCntl).

A device dispatch function can be structured as follows:

```
void* MyDeviceDispatch(VDK_DispatchCode inCode,
                      VDK_DispatchUnion inData)
{
    switch(inCode)
    {
        case VDK_kDD_Init:
            /* Init the device */
        case VDK_kDD_Activate:
            /* Get more data ready for the ISR */
```

Device Drivers

```
case VDK_kDD_Open:
    /* A thread wants to open the device... */
    /* Allocate memory and prepare everything else */
case VDK_kDD_Close:
    /* A thread is closing a connection to the device...*/
    /* Free all the memory, and do anything else */
case VDK_kDD_SyncRead:
    /* A thread is reading from the device */
    /* Return an unsigned int of the number of bytes read */
case VDK_kDD_SyncWrite:
    /* A thread is writing to the device */
    /* Return an unsigned int of the number of bytes
       written */
case VDK_kDD_IOCntl:
    /* A thread is performing device specific actions:
default:
    VDK_DispatchThreadError(VDK_kUnknownDeviceCommand);
    return 0;
}
}
```

Each of the different cases in the dispatch function are discussed below.

Init

The device dispatch function is called with the `VDK_kDD_Init` parameter at system boot time. All device-specific data structures and system resources should be set up at this time. The device driver *should not* call any APIs that throw an error or might block. Additionally, the init function is called within a critical region, and the device driver should not push/pop critical regions, or wait on interrupts.

Open or Close

When a thread opens or closes a device with `OpenDevice()` or `CloseDevice()` APIs, the device dispatch function is called with `VDK_kDD_Open` or `VDK_kDD_Close`. The dispatch function is called from the thread domain, so any stack-based variables are local to that thread. If a thread uses global data structures that are used also by other threads, their access should be protected with an unscheduled region. If the global variables are used by

interrupts subroutines, the device driver should protect their access with critical regions.

When a thread calls the dispatch function attempting to open or close a device, the API passes a union to the device dispatch function whose value is defined with the `OpenClose_t` of the `VDK_DispatchUnion`. The `OpenClose_t` has the following properties:

```
struct OpenClose_t
{
    void      **dataH;
    char      *flags; /* used for kDD_Open only */
};
```

`OpenClose_t.dataH`: A thread-specific handle that a device driver can use to allocate any thread-specific resources. For example, a thread can `malloc` space for a structure that describes the state of a thread associated with a device. The pointer to the structure can be stored in the value that is passed to every other dispatch call involving this thread. A device driver can `free` the space when the thread calls `CloseDevice()`.

`OpenClose_t.flags`: When a thread calls `OpenDevice()`, the second parameter passed is the pointer to any device-specific flags that should be passed to the device dispatch function. The flags passed from the thread are here. Note that this part of the union is not used on a call to `CloseDevice()`.

Read or Write

A thread that needs to read or write to a device it has opened calls `SyncRead()` or `SyncWrite()`. The dispatch function is called in the thread domain and on thread's stack. These functions call the device dispatch function with the parameters passed to the API in the `VDK_DispatchUnion`, and the flags `VDK_kDD_SyncRead` or `VDK_kDD_SyncWrite`. The `ReadWrite_t` is described below:

Device Drivers

```
struct ReadWrite_t
{
    void          **dataH;
    VDK::Ticks    timeout;
    unsigned int  dataSize;
    int           *data;
};
```

`ReadWrite_t.dataH`: The thread-specific value passed to the dispatch function when the function has been called by the device opened by the thread. This handle can be used to store a pointer to a thread-specific data structure detailing what state the thread is in while dealing with the device.

`ReadWrite_t.timeout`: The amount of time in ticks that a thread is willing to wait when pending on a semaphore, event, or device flag.

`ReadWrite_t.dataSize`: The amount of data that the thread reads from or writes to the device.

`ReadWrite_t.data`: A pointer to the location that the thread writes the data to (on a read), or reads from (on a write).

Like calls to the device dispatch function for opening and closing, the calls to read and write are not protected with a critical or unscheduled region. If a device driver accesses global data structures during a read or write, the access should be protected with critical or unscheduled regions. See the discussion in [“Device Flags” on page 4-28](#) for more information about regions and pending.

IOCntl

The VDK supplies an interface for threads to control a device's parameters with the `DeviceIOCntl()` API. When a thread calls `DeviceIOCntl()`, the function sets up some parameters and calls the specified device's dispatch function with the value `VDK_kDD_IOCntl` and the `VDK_DispatchUnion` set up as a `IOCntl_t`. The `IOCntl_t` is described below:

```
struct IOCntl_t
{
    void                **dataH;
    VDK_Ticks           timeout;
    int                 command;
    char                *parameters;
};
```

`IOCntl_t.dataH`: The value passed to the dispatch function when the function has been called by the device opened by the thread. This handle can be used to store a pointer to a thread-specific data structure detailing what state the thread is in while dealing with the device.

`IOCntl_t.timeout`: The amount of time in ticks that a thread is willing to wait when pending on a semaphore, event, or device flag.

`IOCntl_t.command`: A device-specific integer (second parameter from the `VDK_DeviceIOCntl` function).

`IOCntl_t.parameters`: A device-specific pointer (third parameter from the `VDK_DeviceIOCntl` function).

Like read/write and open/close, a device dispatch function call for `IOCntl` is not protected by a critical or unscheduled region. If a device accesses global data structures, the device driver should protect them with a critical or an unscheduled region.

Device Drivers

Activate

Often a device driver needs to respond to state changes caused by ISRs. The device dispatch function is called with a value `VDK_kDD_Activate` at some point after an ISR has called the macro `VDK_ISR_ACTIVATE_DEVICE_DRIVER()`. When the ISR calls `VDK_ISR_ACTIVATE_DEVICE_DRIVER()`, a flag is set indicating that a device has been activated, and the low-priority software interrupt is triggered to run (see [“Reschedule ISR” on page 4-34](#)). When the scheduler is entered through the low-priority software interrupt, a device’s dispatch function is called with the `VDK_kDD_Activate` value.

The activate part of a device dispatch function should handle posting signals so that threads waiting on certain device states can continue running. For example, assume that a D/A ISR runs out of data in its buffer. The ISR would call `VDK_ISR_ACTIVATE_DEVICE_DRIVER()` with the `DeviceID` of its device driver. When the device dispatch function is called with the `VDK_kDD_Activate`, the device posts a device flag, semaphore, or sets an event bit that reschedules any threads that are pending.

Since the activate function is run from the low-priority software interrupt, some uncommon circumstances exist. While the dispatch function is executing a `VDK_kDD_Activate` command, it executes in a critical region and runs off the kernel’s stack. Since there are no threads executing, the dispatch function must avoid calling APIs that throw an error or can block.

Device Flags

Device flags are signals associated with device drivers. Like semaphores and events, a thread can pend on device flags. This means that the thread waits until the flag is posted by the device driver during a call to a device driver's dispatch function.

Pending on a Device Flag

When a thread pends on a device flag, unlike with semaphores and events, the thread always blocks. The thread waits until the flag is posted by another call to the device's dispatch function. When the flag is posted, all threads that are pending on the device flag are moved to the ready queue. Since posting a device flag with the `PostDeviceFlag()` API moves an indeterminate number of threads to the ready queue, the call is non-deterministic. For more information about posting device flags, see [“Posting a Device Flag” on page 4-49](#).

The rules for pending on device flags are strict compared to other types of signals. The "stack" of critical regions must be exactly one level deep when a thread pends on a device flag. In other words, with interrupts enabled, call `PushCriticalRegion()` exactly once prior to calling `PendDeviceFlag()` from a thread. The reason for this condition becomes clear if you consider the reason for pending. A thread pends on a device flag when it is waiting for a condition to be set from an ISR. However, you must enter a critical region before examining any condition that may be modified from an ISR to ensure that the value you read is valid. Furthermore, `PendDeviceFlag()` pops the critical region stack once, effectively balancing the earlier call to

Device Drivers

`PushCriticalRegion()`. For example, a typical device driver uses device flags in the following manner:

```
VDK_PushCriticalRegion();
while(should_loop != 0)
{
    /* ... */
    /* access global data structures */
    /* and figure out if we should keep looping */
    /* ... */

    /* Wait for some device state */
    VDK_PendDeviceFlag();

    /* Must re-enter the critical region */
    VDK_PushCriticalRegion();
}
VDK_PopCriticalRegion();
```

Figure 4-12 illustrates the process of pending on a device flag.

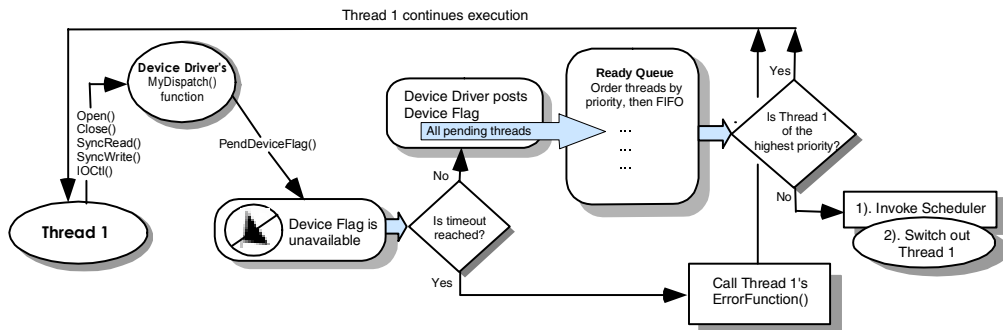


Figure 4-12. Pending on a Device Flag

Posting a Device Flag

Like semaphores, a device flag can be posted. A device dispatch function posts a device flag with a call to `PostDeviceFlag()`. Unlike semaphores, the call moves *all* threads pending on the device flag to the ready queue and continues execution. Once `PostDeviceFlag()` returns, subsequent calls to `PendDeviceFlag()` cause the thread to block (as before).

Note that the `PostDeviceFlag()` API does not throw any errors. The reason for this is that the API function is called typically from the dispatch function when the dispatch function has been called with `VDK_kDD_Activate`. This is because the device dispatch function operates on the kernel's stack when it is called with `VDK_kDD_Activate` rather than on the stack of a thread.

Figure 4-13 illustrates the process of posting a device flag.

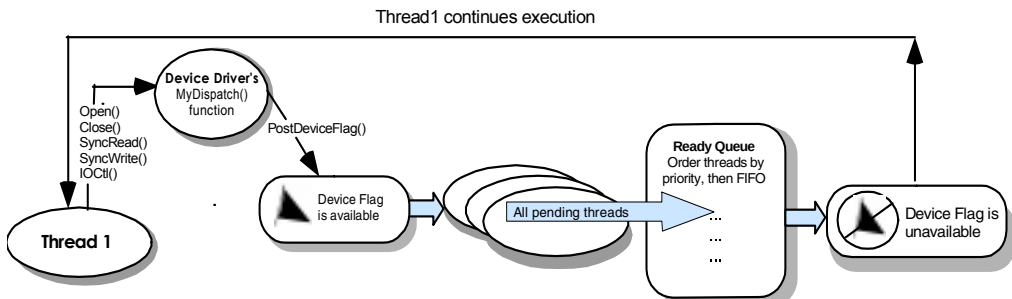


Figure 4-13. Posting a Device Flag

General Notes

Keep the following tips in mind while writing device drivers. Although many of these topics also apply to threads, they deserve special mention with respect to device drivers.

Variables

Device drivers and ISRs are closely linked. Since ISRs and the dispatch function access the same variables, the easiest way is to declare the variables in the C/C++ device driver routine, and to access them as `extern` from within the assembly ISR. When declaring the variables in the C/C++ source file, you must declare the variables as `volatile` to prevent compiler optimizations impeding assembly language access to the variables. Additionally, care must be taken in the ISR to mangle the name correctly.

Notes on Critical/Unscheduled Regions

Since many of the data structures and variables associated with a device driver are shared between multiple threads and ISRs, access to them must be protected within critical regions and unscheduled regions. Critical regions keep ISRs from modifying data structures unexpectedly, and unscheduled regions prevent other threads from modifying data structures.

Care must be taken when pending on flags in the right regions. Since device flags are the only signals that can be pending on from within a critical region and device flags pop the critical region, careful region balancing must be adhered to. Take advantage of the errors that are thrown by the APIs only when instrumentation is enabled to detect mismatches. For more information about instrumentation, refer to [page 3-8](#).